# Chapter 13: Copy Control

We'll cover the basics of the copy constructor, the copy-assignment operator, and destructor. The move constructor is covered last.

## Copy Constructor

The copy constructor has a signature like this:

```
class MyClass {
    int att = 0;
public:
    MyClass(const int i): att(i){}
    MyClass(const MyClass &rhs): att(rhs.att){
    }
    int getAtt(){ return this->att; }
};

int main(){
  MyClass a(3);
  cout << a.getAtt() << endl;
  MyClass b(a); // or MyClass b = a;
  cout << b.getAtt() << endl;
}
```

A copy constructor is invoked whenever a new object is created and initialized through an existing object of the same kind. This happens in several situations. The most obvious situation is **when you explicitly initialize a new object from an existing object**.

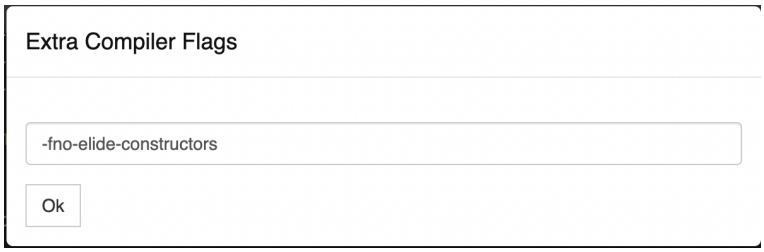
```
string motto = "motto";

// calls string(const string &)
string ditto(motto);
string metoo = motto;
string also = string(motto);
string* pString = new string(motto);
```

In C++, a Copy Constructor may be called in the following cases[1]:

1. When an object of the class is returned by value.
2. When an object of the class is passed (to a function) by value as an argument.
3. When an object is constructed based on another object of the same class.
4. When the compiler generates a temporary object.

[1] Content from: https://www.geeksforgeeks.org/when-is-a-copy-constructor-called-in-cpp/

To see the effects of the copy constructor more clearly, you need to disable the compiler's return value optimization (RVO)[2]: -fno-elide-constructors




With your own class, you can easily check whether the copy constructor has been called.

```
#include <iostream>
using namespace std;

class MyClass{
public:
    MyClass() = default ;
    MyClass(int a):i(a){}
    MyClass(const MyClass& rhs):i(rhs.i){
        cout << "MyClass::MyClass(const MyClass& rhs): " << ref++
             << " time(s)" << endl;
    }
    int i = 0;
    static int ref;
};

int MyClass::ref = 1;

MyClass foo(MyClass m) {
    MyClass m3(m);
    cout << "Before returning from foo" << endl;
    return m3;
}

int main(){
    MyClass m1(7);
    MyClass m2 = m1;
    cout << "Before calling foo" << endl;
    MyClass m4 = foo(m1);
    return 0;
}
```

Outputs:

```
MyClass::MyClass(const MyClass& rhs): 1 time(s)
Before calling foo
MyClass::MyClass(const MyClass& rhs): 2 time(s)
MyClass::MyClass(const MyClass& rhs): 3 time(s)
Before returning from foo
MyClass::MyClass(const MyClass& rhs): 4 time(s)
MyClass::MyClass(const MyClass& rhs): 5 time(s)
```

[2] More info please refer to: https://www.geeksforgeeks.org/copy-elision-in-c/

The fact that the copy constructor is invoked for **non-reference parameters** explains why the copy constructor's own parameter must be a reference. If not, we will see recursion.

> [Bad recursion]: To call the copy constructor, we'd need to use the copy constructor to copy the argument, but to copy the argument, we'd need to call the copy constructor, and so on indefinitely.

**Remark:** In some of these cases, an object is immediately destroyed after it is copied. It is not efficient. We will talk about a better way: **move** constructor at the end of this lecture.

When no copy constructor is defined, the compiler synthesizes one for us. The **synthesized copy constructor member-wise copies** each non-*static* member in turn from the given object into the one being created. Problematic for data members with pointer and dynamic memory.

```
#include <iostream>
using namespace std;

class MyClass{
public:
    MyClass() = default ;
    MyClass(int a, int size, int val):i(a), as(size){
      ia = new int[as];
      for(int j = 0; j < as; ++j)
        ia[j] = val;
    }
    //MyClass(const MyClass& rhs):i(rhs.i),as(rhs.as), ia(rhs.ia){
    // same as synthesized copy constructor
    //} // ia is shallow copied => only pointer copied not memory
    int i = 0;
    int as = 0;
    int* ia = nullptr; // this could be problematic when copying
};

int main(){
    MyClass m1(7, 5, -1);
    MyClass m2 = m1;

    m1.ia[0] = 100;

    cout << m1.ia[0] << endl;
    cout << m2.ia[0] << endl; // same copy, point to same memory

    delete m1.ia;
    //delete m2.ia; // error because pointers lead to same memory

    return 0;
}
```

A better way to handle this is to create separate copy of resource (deep copy) for each object.

```
#include <iostream>
using namespace std;

class MyClass{
public:
    MyClass() = default ;
    MyClass(int a, int size, int val):i(a), as(size){
      ia = new int[as];
      for(int j = 0; j < size; ++j)
        ia[j] = val;
    }
    MyClass(const MyClass& rhs):i(rhs.i), as(rhs.as){
      ia = new int[as]; // allocate new memory for new object
      for(int j = 0; j < as; ++j)
        ia[j] = rhs.ia[j]; // deep copy
    }
    int i = 0;
    int as = 0;
    int* ia = nullptr;
};

int main(){
    MyClass m1(7, 5, -1);
    MyClass m2 = m1;

    m1.ia[0] = 100;

    cout << m1.ia[0] << endl;
    cout << m2.ia[0] << endl; // separate copy

    delete m1.ia;
    delete m2.ia; // no error

    return 0;
}
```

When we make a copy constructor private in a class, objects of that class become non-copyable. If this is what you want, do it.

## Copy Assignment Operator

The copy assignment operator has a signature like this:

```
  class MyClass {
  public:
    ...
    MyClass & operator=(const MyClass &rhs);
    ...
  }
```

```
  MyClass a, b;
```

```
...
b = a;   // Same as b.operator=(a);
```

Notice that the = operator takes a `const`-reference to the right-hand side of the assignment. We don't want to change that value; **we only want to change what's on the left-hand side.**

Also, a reference is returned by the assignment operator. This is to allow **operator chaining**.

Now, one more very important point about the assignment operator: YOU MUST CHECK FOR SELF-ASSIGNMENT!

This is especially important **when your class does its own memory allocation**. The typical sequence of operations within an assignment operator is like this:

```
MyClass& MyClass::operator=(const MyClass &rhs) {
  // 1.  Deallocate any memory that MyClass is using internally
  // 2.  Allocate some new memory to hold the contents of rhs
  // 3.  Copy the values from rhs into this instance
  // 4.  Return *this
}
```

Now, what happens when you do something like this:

```
MyClass mc;
...
mc = mc;     // self assignment: bad news!
```

Because `mc` is on the left-hand side and on the right-hand side, the first thing that happens is that `mc` releases any memory it holds internally. But this is where the values were going to be copied from, since `mc` is also on the right-hand side!

The safer way is to CHECK FOR SELF-ASSIGNMENT:

```
MyClass& MyClass::operator=(const MyClass &rhs) {
  if (this == &rhs)// Same object?
    return *this;// skip assignment
  else {
  ... // Deallocate, allocate new space, copy values...
  }

  return *this;
}
```

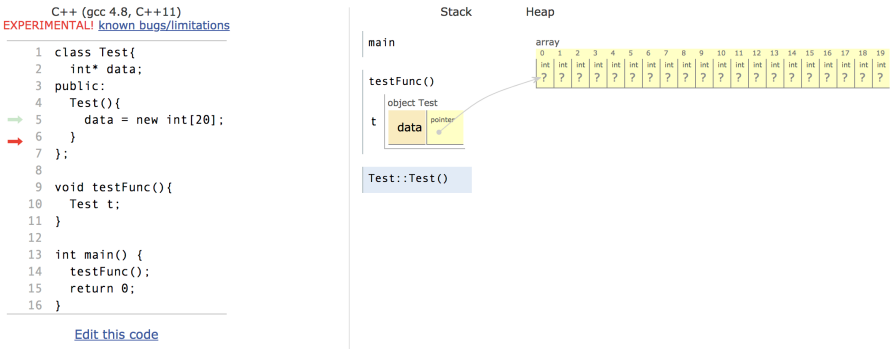
Or, you can simplify this a bit by doing:

```
MyClass& MyClass::operator=(const MyClass &rhs) {
  if (this != &rhs) {
    ... // Deallocate, allocate new space, copy values...
  }

  return *this;
}
```

In summary, the guidelines for the assignment operator are:

1. Take a `const`-reference for the argument (the right-hand side of the assignment).
2. Return a reference to the left-hand side, to support safe and reasonable operator chaining. (Do this by returning `*this`.)
3. Check for self-assignment, by comparing the pointers (`this` to `&rhs`).

## Destructor




One rule of thumb to use when you decide whether a class needs to define its own versions of the copy-control members is to decide first whether the class needs a **destructor**: need to free up heap memory? Did you use **new** in the constructor?

## The Rule of Three

Often, **the need for a destructor** is more obvious than the need for the copy constructor or copy assignment operator. If the class needs a destructor, it almost surely needs a copy constructor and assignment operator. For example:

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
private:
    std::string *ps;
    int i;
};
```

It is obvious that we will need our own destructor to clean up the memory. What may be less clear—but what our rule of thumb tells us—is that **`HasPtr` also needs a copy constructor and copy assignment operator.**

If we gave `HasPtr` a destructor but used the synthesized versions of the copy constructor and copy-assignment operator:

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
       ps(new std::string(s)), i(0) { }
    ~HasPtr(){delete ps;}
private:
    std::string *ps;
    int i;
};
```

The memory allocated in the constructor will be freed when a `HasPtr` object is destroyed. Unfortunately, **we have introduced a serious bug!** This class uses the synthesized copy and assignment. Those functions copy the pointer member: multiple `HasPtr` objects point to the same memory:

```
HasPtr f(HasPtr hp){ // HasPtr passed by value, so it is copied
    HasPtr ret = hp; // copies the given HasPtr
    // process ret
    return ret; // ret and hp are destroyed and memory released
}
```

When `f` returns, the destructor will delete the pointer member in `ret` and in `hp`. But these objects contain the same pointer value. This code will delete that pointer twice, which is an error and **its behavior is undefined**.

In addition, the caller of `f` may still be using the object that was passed to `f`:

```
HasPtr p("some values");
f(p); // when f completes, the memory to which p.ps points is
      // freed, now p points to invalid memory!
```

Let us observe the above behavior into a complete program:

Example: Missing Copy Constructor

```
#include <string>
#include <iostream>
```

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
       ps(new std::string(s)), i(0) { }
    ~HasPtr(){delete ps;}
    void printString() {std::cout << *ps;}
private:
    std::string *ps;
    int i;
};

HasPtr f(HasPtr hp) {// HasPtr passed by value, so it is copied
    HasPtr ret = hp; // copies the given HasPtr
    // process ret
    return ret; // ret and hp are destroyed
}

using namespace std;

int main(){
    HasPtr p("some values");
    p.printString();
    f(p); // when f completes, the memory p.ps points is freed
    cout << endl;
    cout << "After f: " << endl;
    p.printString();
    cout << endl;
    HasPtr q(p); // now both p and q point to invalid memory!
    cout << "After q(p): " << endl;
    p.printString();
    cout << endl;
    q.printString();
    cout << endl;
}
```

We will get error!

If we add the proper copy constructor into the code:

```
    HasPtr(const HasPtr& orig)  {
       ps = new std::string();
       *ps = *orig.ps;
       i = orig.i;
    }
```

We will then get the expected results:

```
some values
After f:
some values
After q(p):
some values
some values
```

## The move constructor[3]

The *move constructor* moves resources in the heap, i.e., dynamic memory, to the new object.

Unlike the copy constructor that copies the data of the existing object and assigning it to the new object, the move constructor just makes the pointer of the declared object to point to the data of the temporary object and nulls out the pointer of the temporary objects.

The move constructor prevents unnecessarily copying of data in dynamic memory.

Work of move constructor looks a bit like default member-wise copy constructor but in this case, it nulls out the pointer of the temporary object, preventing more than one object to point to same memory location.

Syntax of the Move Constructor:

```
Object_name(Object_name&& obj)
   : data( obj.data ) {
   // obj.data is pointer to object(s) in heap
   // Nulling out the pointer to the temporary data
   obj.data = nullptr;
}
```

What happens when destructing the `obj`? When `delete` on pointer to `nullptr` (or `null`), the "*default deallocation functions are guaranteed to do nothing*[4]."

```
#include <iostream>

using namespace std;

int main(){
    cout<< "Hello World" << endl;

    int* i = new int(1);
    cout<< *i << endl;;
    delete i;
    cout << "after first delete" << endl;
    delete i;
```

[3] The move constructor https://www.geeksforgeeks.org/move-constructors-in-c-with-examples/
[4] Delete on null https://en.cppreference.com/w/cpp/language/delete

```
    cout << "after second delete" << endl;
    return 0;
}
```

For temporary objects, they are destroyed as the last step in evaluating the full-expression (that contains the point where they were created)[5]. In other words, once they are not used, they are destroyed.

Example:

```
#include <iostream>
#include <vector>

using namespace std;

class OOPMoveDemo {
private:
    int* data; // should point to dynamic memory (heap)

public:

    OOPMoveDemo (int d){
        data = new int; // new allocates on heap
        *data = d;
        cout << "Constructor is called for "
             << d << endl;
    };

    // Copy Constructor
    OOPMoveDemo (const OOPMoveDemo & source)
        : OOPMoveDemo ( *source.data ){
        cout << "Copy Constructor is called -"
             << "Deep copy for "
             << *source.data
             << endl;
    }

    // Move Constructor
    OOPMoveDemo (OOPMoveDemo && source)
        : data( source.data ){
        cout << "Move Constructor for "
             << *source.data << endl;
        source.data = nullptr;
    }

    // Destructor
    ~OOPMoveDemo (){
        if (data != nullptr)
            cout << "Destructor is called for "
                 << *data << endl;
        else
```

[5] Temporary object lifetime https://en.cppreference.com/w/cpp/language/lifetime

```
            cout << "Destructor is called"
                << " for nullptr "
                << endl;

        // Free up the memory assigned to
        // The data member of the object
        delete data;
    }
};

int main(){
    vector<OOPMoveDemo> vec;
    vec.reserve(2); // reserve space for 2

    // create temporary object with val 1
    cout << "construct obj with val 1:" << endl;
    OOPMoveDemo(1); // temporary object
    cout << endl;

    // Inserting Object of Move Class
    cout << "construct obj with val 10 and push_back to vec:" <<
endl;
    vec.push_back(OOPMoveDemo(10));
    cout << endl;

    cout << "construct obj with val 20 and push_back to vec:" <<
endl;
    vec.push_back(OOPMoveDemo(20));
    cout << endl;

    cout << "Program ending, resources will be freed:" << endl;
    return 0;
}
```

Output:

```
construct obj with val 1:
Constructor is called for 1
Destructor is called for 1

construct obj with val 10 and push_back to vec:
Constructor is called for 10
Move Constructor for 10
Destructor is called for nullptr

construct obj with val 20 and push_back to vec:
Constructor is called for 20
Move Constructor for 20
Destructor is called for nullptr

Program ending, resources will be freed:
Destructor is called for 10
Destructor is called for 20
```

If you comment out the move constructor:

```
construct obj with val 1:
Constructor is called for 1
Destructor is called for 1

construct obj with val 10 and push_back to vec:
Constructor is called for 10
Constructor is called for 10
Copy Constructor is called -Deep copy for 10
Destructor is called for 10

construct obj with val 20 and push_back to vec:
Constructor is called for 20
Constructor is called for 20
Copy Constructor is called -Deep copy for 20
Destructor is called for 20

Program ending, resources will be freed:
Destructor is called for 10
Destructor is called for 20
```

You can see the copy constructor was called many times more, more heap is used, and immediately heap memory released for the temporary object. (both object with val 10 and 20 had these wastes.)

**Remark:** The ability to **move** rather than **copy** an object could be more efficient. In some of these cases, an object is immediately destroyed after it is copied. It is not efficient. Moving, rather than copying, the object can provide a significant performance boost.

## Rule of five

The presence of a user-defined **destructor**, **copy-constructor**, or **copy-assignment operator** prevents implicit definition of the **move constructor** and the **move assignment operator**. Any class for which move semantics are desirable, has to declare all **five** special member functions.

## Move assignment operator

```
#include <iostream>
#include <vector>

using namespace std;

class OOPMoveDemo {
private:
    int* data; // should point to dynamic memory (heap)
```

```
public:
   OOPMoveDemo ():data(nullptr){
      cout << "Default Constructor is called" << endl;
   };
    OOPMoveDemo (int d){
        data = new int; // new allocates on heap
        *data = d;
        cout << "Constructor is called for "
             << d << endl;
    };

    // Copy Constructor
    OOPMoveDemo (const OOPMoveDemo & source)
        : OOPMoveDemo ( *source.data ){
        cout << "Copy Constructor is called -"
             << "Deep copy for "
             << *source.data
             << endl;
    }
    // Copy Assignment Operator
    OOPMoveDemo& operator=(const OOPMoveDemo & source){
        if(this != &source){
            delete this->data;
            this->data = new int(*source.data);
        }
        cout << "Copy Assignment for "
             << *source.data << endl;
        return *this;
    }
    // Move Constructor
    OOPMoveDemo (OOPMoveDemo && source)
        : data( source.data ){
        cout << "Move Constructor for "
             << *source.data << endl;
        source.data = nullptr;
    }

    // Move Assignment Operator
    OOPMoveDemo& operator=(OOPMoveDemo && source){
        if(this != &source){
            delete this->data;
            this->data = source.data;
            source.data = nullptr;
        }
        cout << "Move Assignment for "
             << *this->data << endl;
        return *this;
    }

    // Destructor
    ~OOPMoveDemo (){
        if (data != nullptr)
            cout << "Destructor is called for "
                 << *data << endl;
        else
            cout << "Destructor is called"
```

```
                << " for nullptr "
                << endl;

        delete data;
    }
};

OOPMoveDemo create(){
    return OOPMoveDemo(1);
}

int main(){
    OOPMoveDemo m;
    cout << "before call to create()" << endl;
    m = create();
    cout << "before returning from main" << endl;

    return 0;
}
```

```
Default Constructor is called
before call to create()
Constructor is called for 1
Move Assignment for 1
Destructor is called for nullptr
before returning from main
Destructor is called for 1
```

If both copy assignment and move assignment operators are provided, overload resolution selects[6]:

- the move assignment if the argument is a nameless temporary or a result of `std::move()`, and selects
- the copy assignment if the argument is an lvalue (named object or a function/operator returning lvalue reference).

[6] Move assignment operator https://en.cppreference.com/w/cpp/language/move_assignment

# Appendix

## Assignment Operator's return type: reference or not

The following expression

```
int a, b, c, d, e;
a = b = c = d = e = 42;
```

is interpreted by the compiler as:

```
a = (b = (c = (d = (e = 42))));
```

Assignments are **right-associative**. The last assignment operation is evaluated first, and is propagated leftward through the series of assignments.

Now, in order to support **operator chaining**, the assignment operator return a reference to the left-hand side of the assignment.

**Q:** If we return an object rather than a reference, we know a new object will be created. Will this affect the results of operator chaining `a=b=c` if we return an object instead of a reference? Will it affect the results of operator chaining `(a=b)=c`?

**A:**

Case 1: the results are ok

```
a = b = c
a = (b = c) → b is now equal to c
a = tmp_b → a is now equal to tmp_b equal to c
```

Case 2: the results are not ok (but can be compiled)

```
(a = b) = c → a is equal to b
tmp_a = c → tmp_a is equal to c
```

**Remark:** for simple chain assignment operation `a = b = c`, both return non-`const` reference or non-`const` object works but we prefer non-`const` reference for the reason of efficiency.

Quick Concept Check: Let us now design a program to test the above statements

```
#include <iostream>
using namespace std;

class MyClass {
```

```
public:
    MyClass(int i):data(i){}
    // act like synthesis assignment operator
    MyClass& operator=(const MyClass &rhs) {
        if (this != &rhs) data = rhs.data;
        return *this;
    }
    int print(){return data;}
private:
    int data;
};

int main(){
    MyClass a(1), b(2), c(3);
    a = b = c;
    cout << a.print() << " " << b.print() << " " << c.print() <<
endl;
    MyClass d(1), e(2), f(3);
    (d=e)=f;
    cout << d.print() << " " << e.print() << " " << f.print() <<
endl;
    return 0;
}
```

**Q:** what are the outputs?

**A:**

```
3, 3, 3
3, 2, 3
```

**Q:** what are the outputs if we return an object? (change code above)

```
MyClass operator=(const MyClass &rhs) {
```

**A:**

```
3, 3, 3
2, 2, 3
```